Shifting Testing Left

CS 3250
Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch. 4]

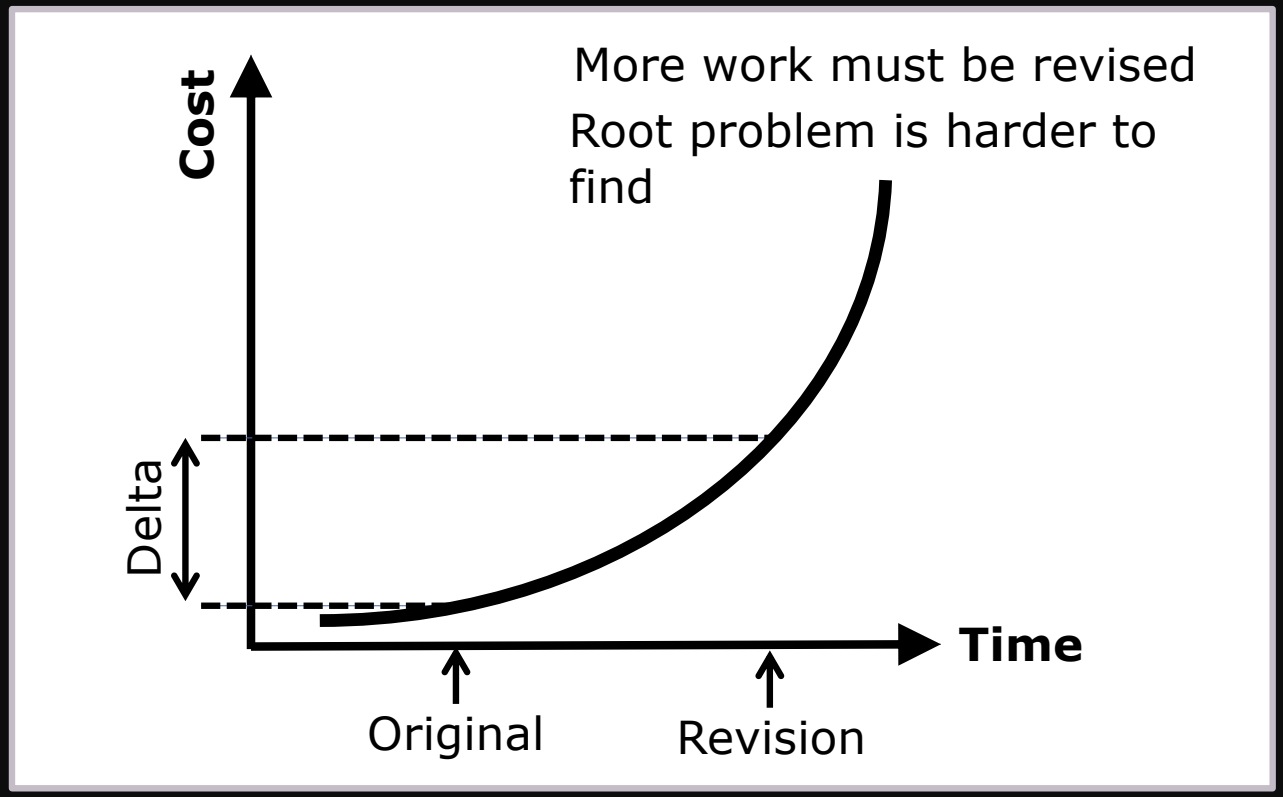
Software Crisis

- Programmers just programmed
  - Result was ad hoc structure
  - Code eventually became hard to maintain
  - Managing complexity was challenging
- The tendency towards irreducible number of errors
- Most software development faced
  - Overdue schedule
  - Exceeding initial budget
  - Inadequate software quality
  - High software maintenance cost

Traditional Cost-of-Change Curve

Traditional software development methods

- Focus: extensive modeling and upfront analysis
- Goal: reveal problems and changes as early as possible



[AO, p.55]

Traditional Assumptions

1. Modeling and analysis can identify potential problems and changes early in development
2. Saving implied by the cost-of-change curve justify the cost of modeling and analysis over the life of the project

- These assumptions are true if the requirements are always complete and current
- In reality, customers keep changing their mind
- Changes reflect the requirements

Increased Emphasis on Testing

If high-quality testing is not centrally and deeply embedded in your development process, your project is at high risk for failure

- Fail in the technical sense
  - Lose control of what the code actually does
- Fail in the business sense
  - Your competitors roll out better functionality faster

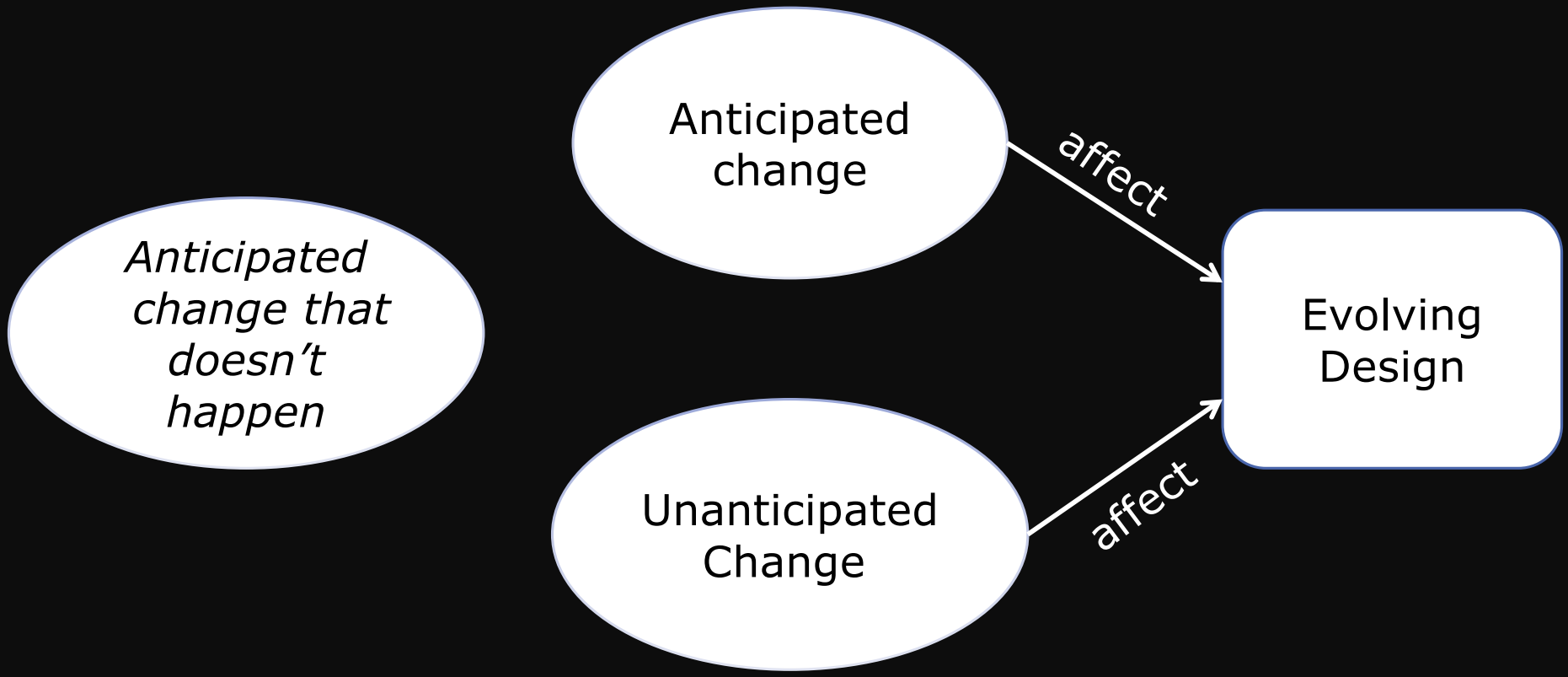
One obvious solution is Agile

Agile Methods

- Both traditional assumptions are invalid for many current software projects
  - Software engineers are not good at developing requirements
  - We do not anticipate many changes
  - Many of the changes we do anticipate are not needed
- Requirements (and other non-executable artifacts) tend to go out of date very quickly
  - We rarely (or almost never) update them
  - Many current software projects change continuously
- Agile methods start small – with some behaviors and specific tests – and then evolve over time

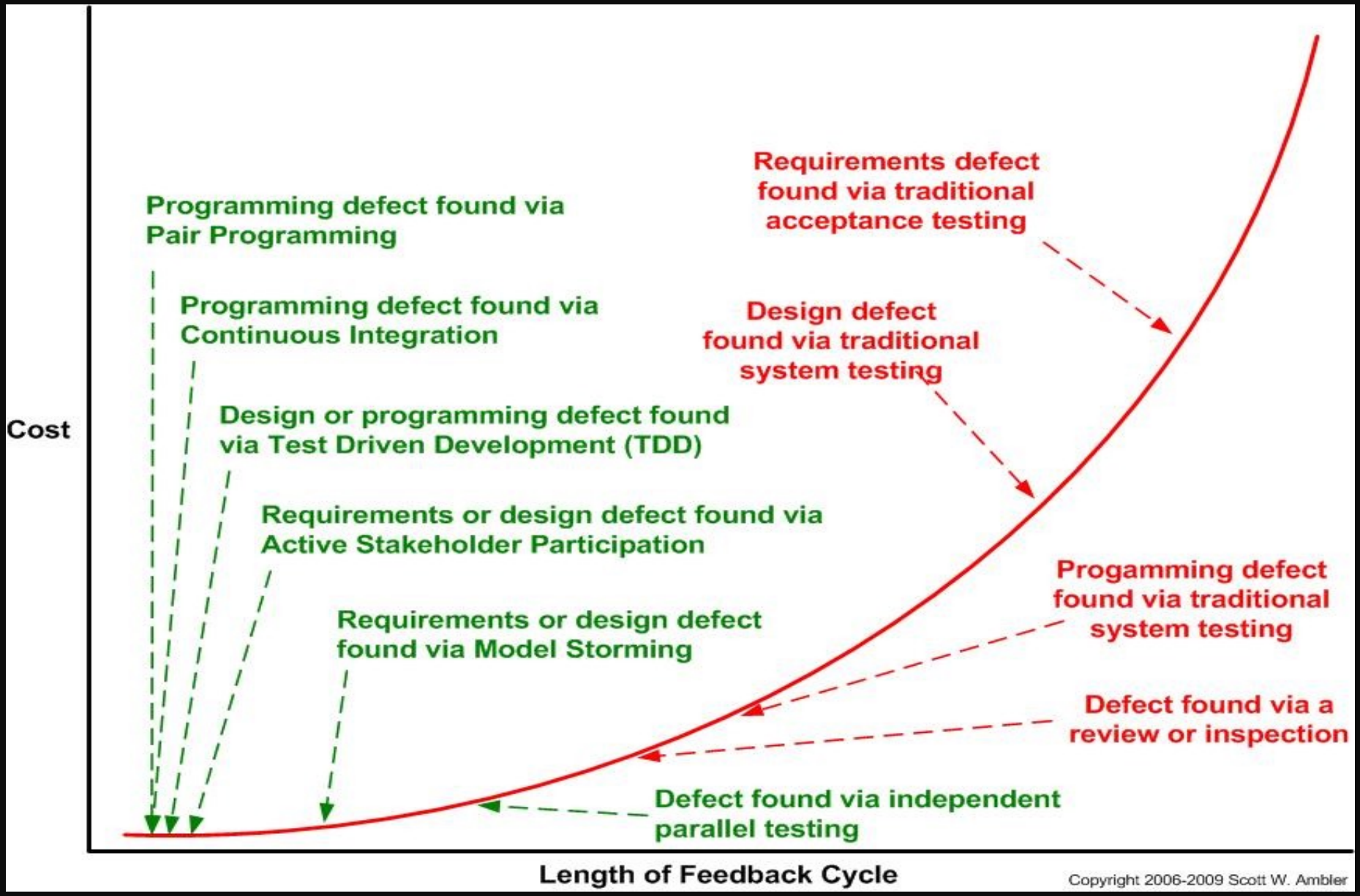
Iterative development to accommodating customers' demands and expectations

Supporting Evolutionary Design



[AO, Agile Test]

Defect Discovery: Traditional vs. Agile



Managing the Cost Curve

- Test harness as guardian
  - (Near) Instant feedback on changes (or mistakes)
    - An hour? Ten minutes? Less?
  - Something is executable from the very beginning
- Role of continuous integration
  - Effective communication mechanisms
- De-emphasize non executable artifacts
  - If it doesn't execute, it's not checkable
- Avoid anticipating future needs
  - YAGNI: You Ain't Gonna Need It

Test Harness as Guardian

- What is correctness?
  - Traditional: universal
  - Agile: existential
- Limit view of correctness
  - Traditional: define all correct behavior completely at the beginning
  - Agile: define correctness of some behavior with specific tests
    - If the software behaves correctly on the tests, it is correct

Even as the software (including the test cases) evolve, the correctness of the system at any single point in time is subject to immediate verification by running the test set.

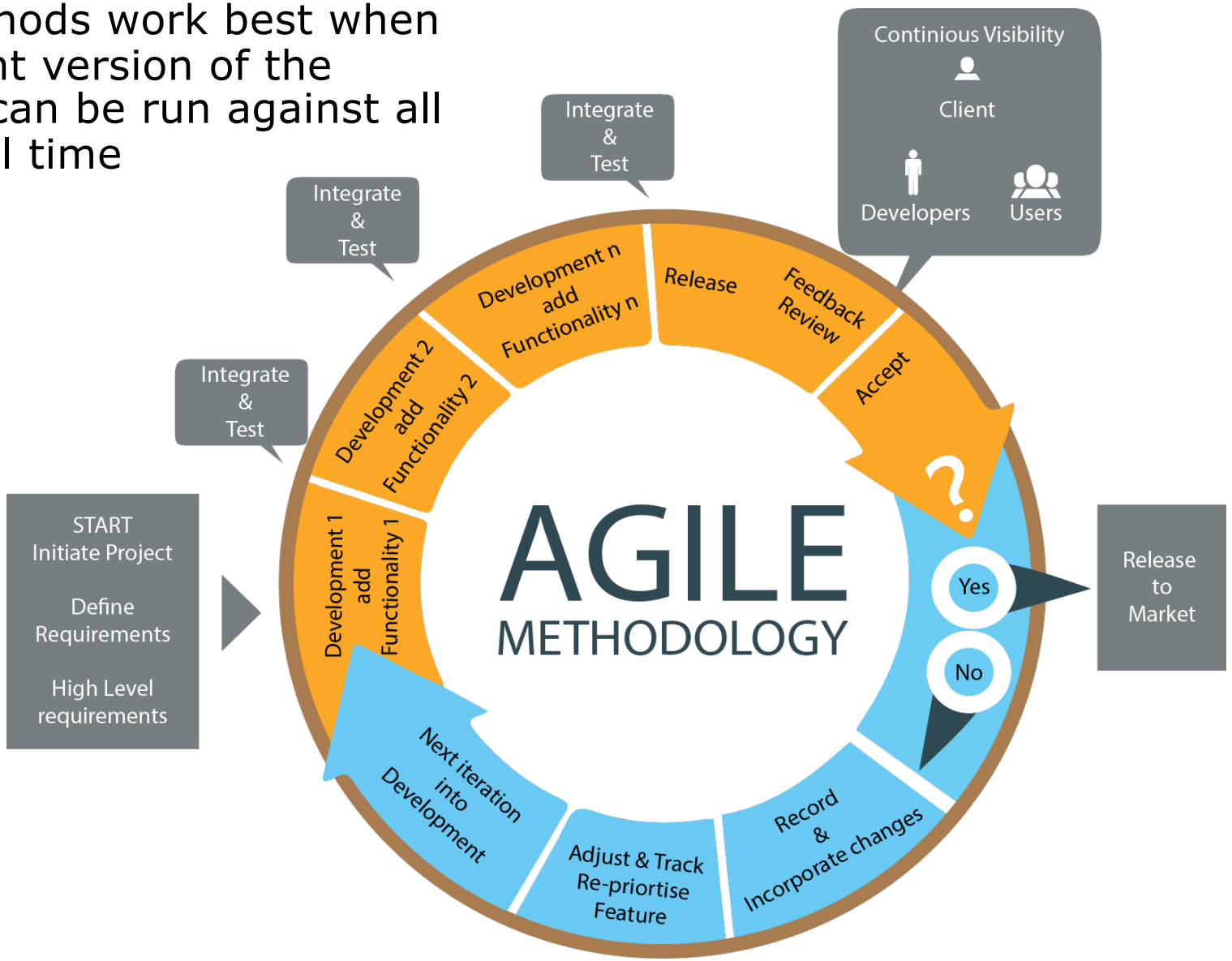
Test Harness Verify Correctness

- Tests must be automated
- Every test must include a test oracle (mechanism that can evaluate whether that test passes or fails)
- Tests (executable artifacts) replace the requirements (non-executable artifacts)
- Tests must be high quality and must run quickly
- Tests must be run every time changes are made to the software

Test harness runs all automated tests efficiently and reports results to the developers

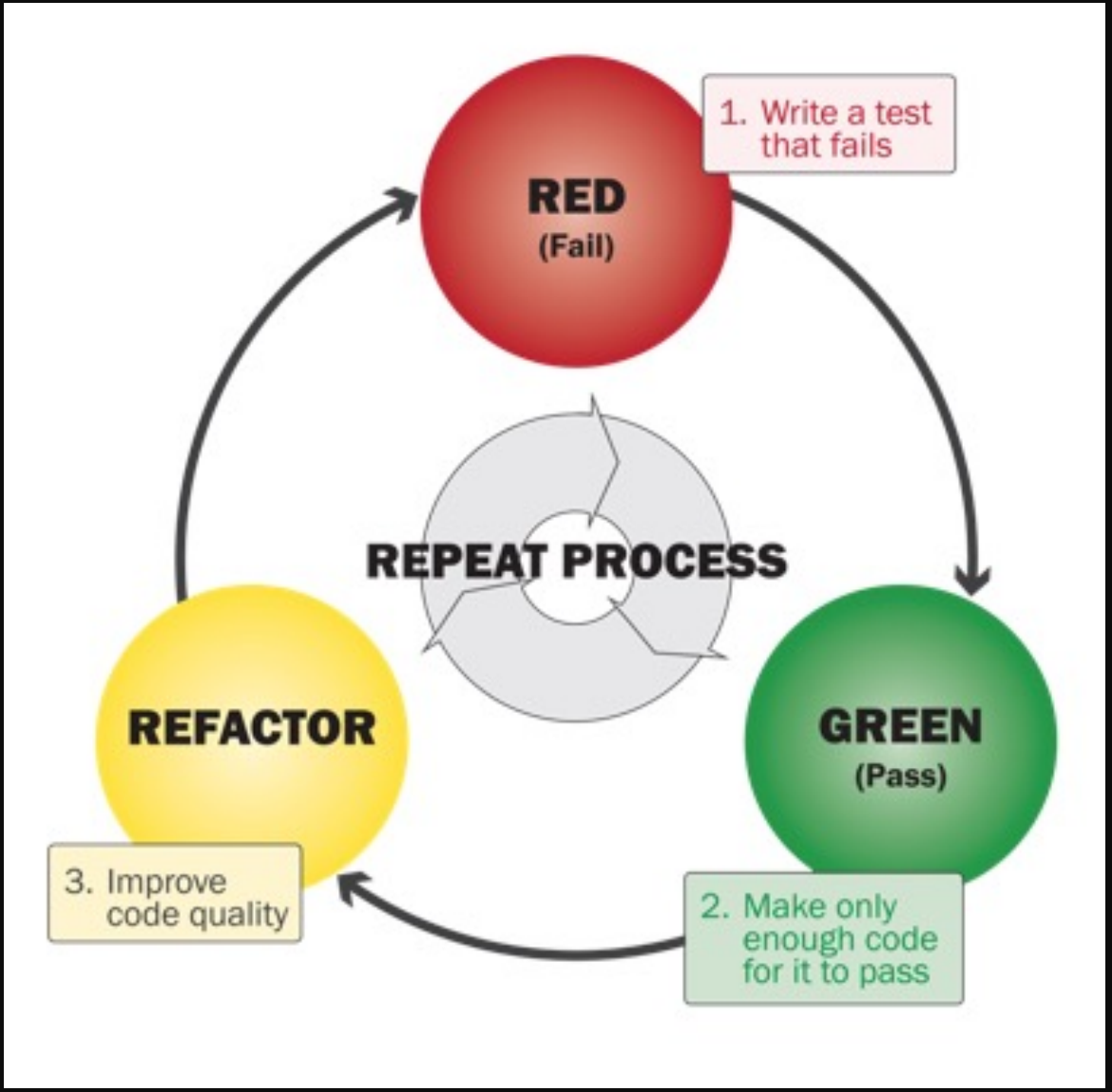
Testing as Central Activity

Agile methods work best when the current version of the software can be run against all tests at all time



[image from http://www.twilightsoftwares.com]

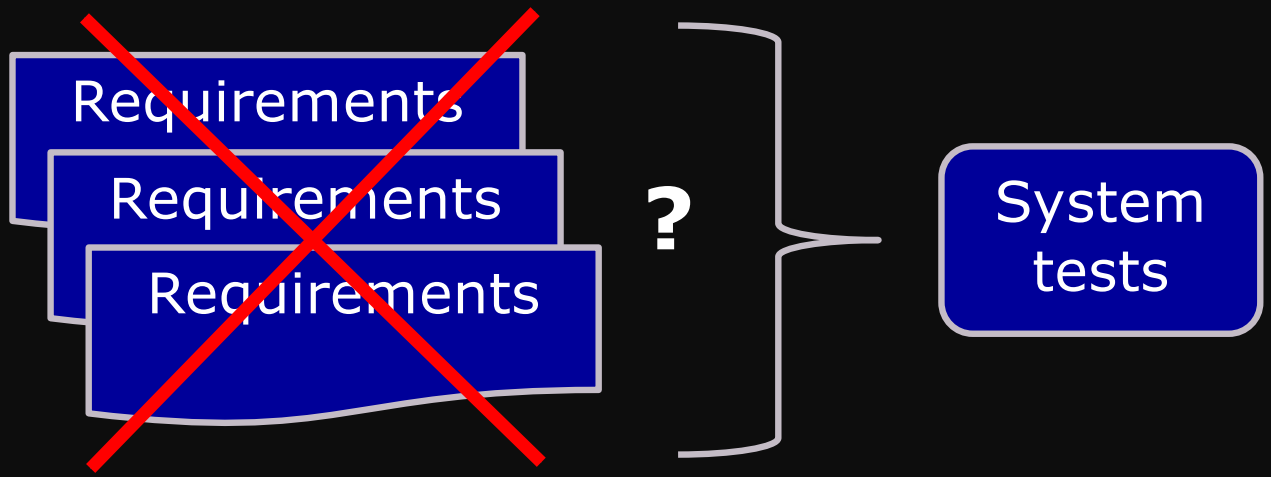
Testing as Central Activity: TDD



[More TDD and exercise .. Later]

System Tests in Agile Methods

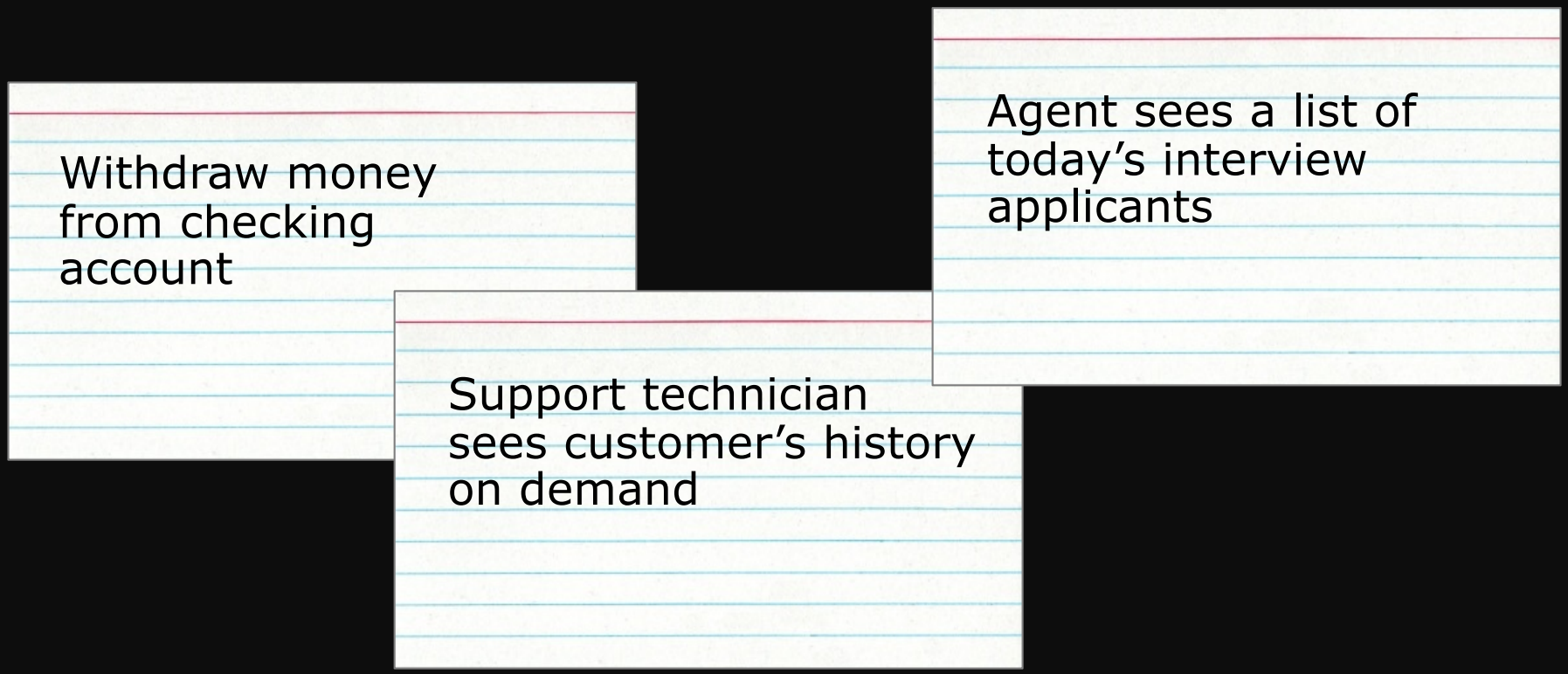
- Traditional testers often design system tests from requirements



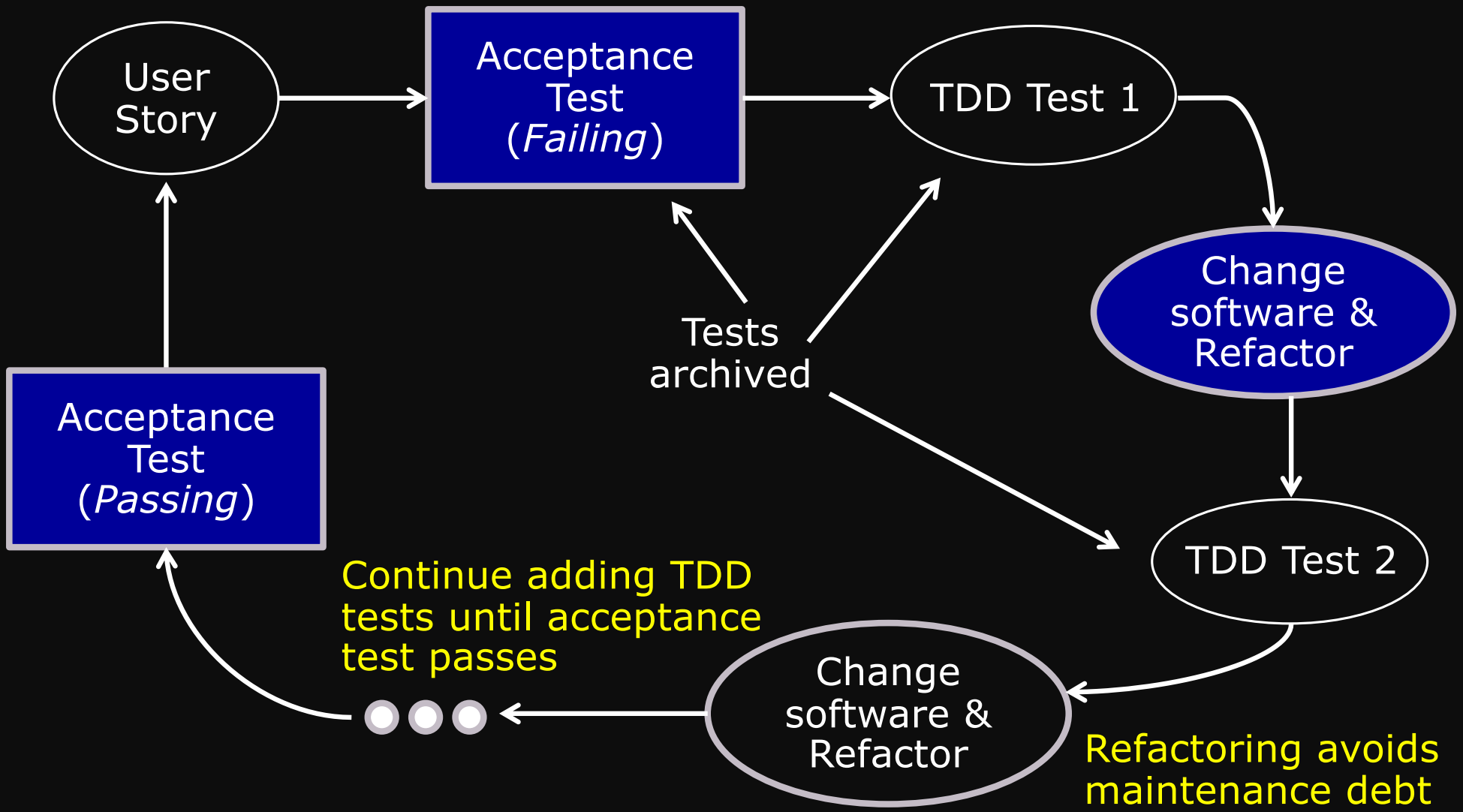
- What if there are no traditional requirement documents?
- What if the traditional requirement documents are outdated?

User Stories

- A few sentences that captures what a user will do with the software
  - In the language of the end user
  - Usually small in scale with few details
  - Not archived



Acceptance Tests in Agile Methods



[AO, p.60]

Continuous Integration

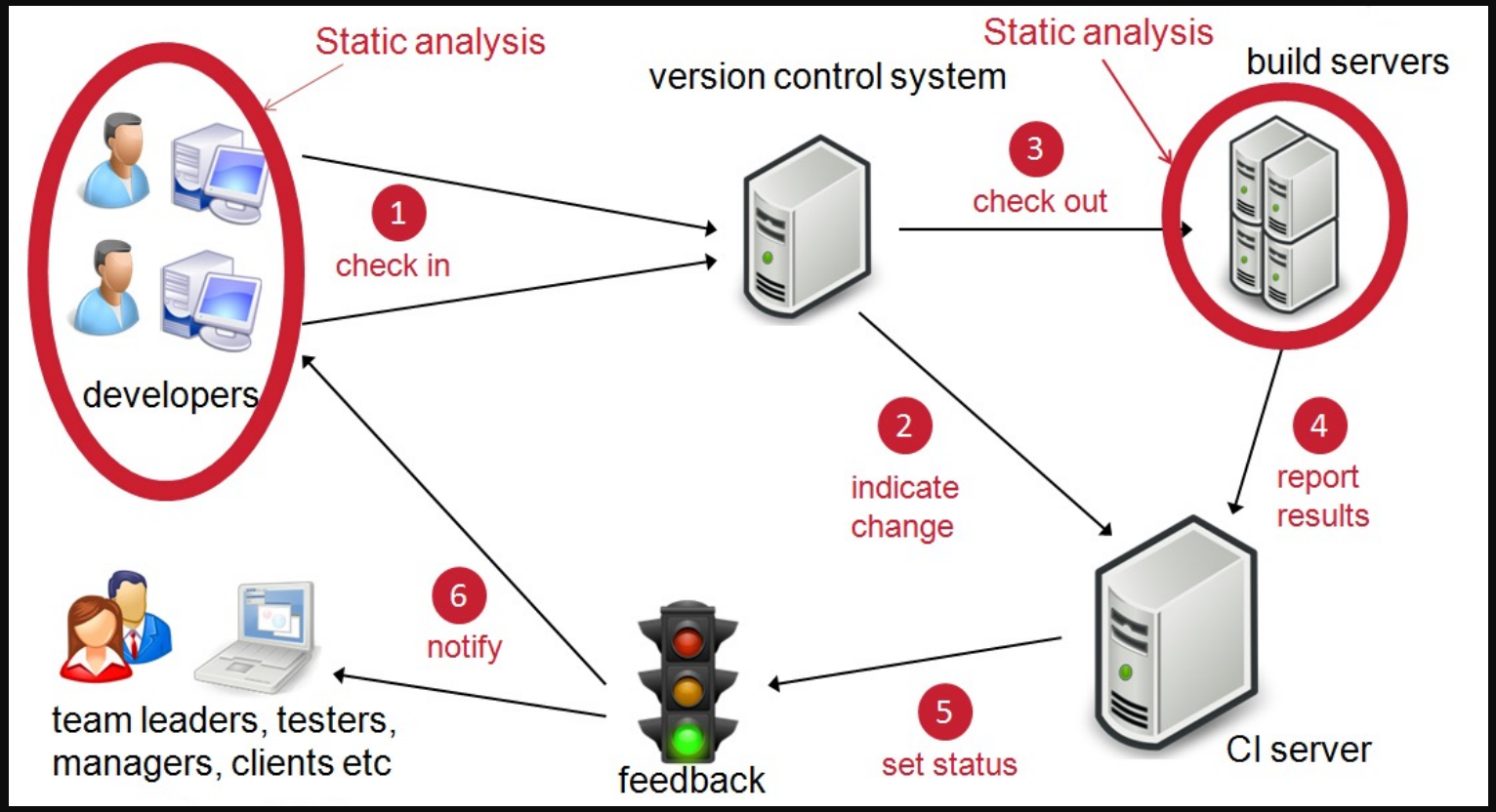
- Architecture-based software development practice
  - Flexible and possible to deliver high-quality software in extremely short timeframes
- Goal: never break the build
  - Test each piece of code thoroughly
  - Ensure that untested or broken code does not get committed
  - Implement strict version control policies
- Team members integrate their work frequently, leading to multiple integrations per day
- Each integration is verified by an automated build and test to detect integration errors as quickly as possible

Build, test, integrate – frequently incrementally, continuously

Continuous Integration (CI)

Developers submit the new code/changes to a central code repository

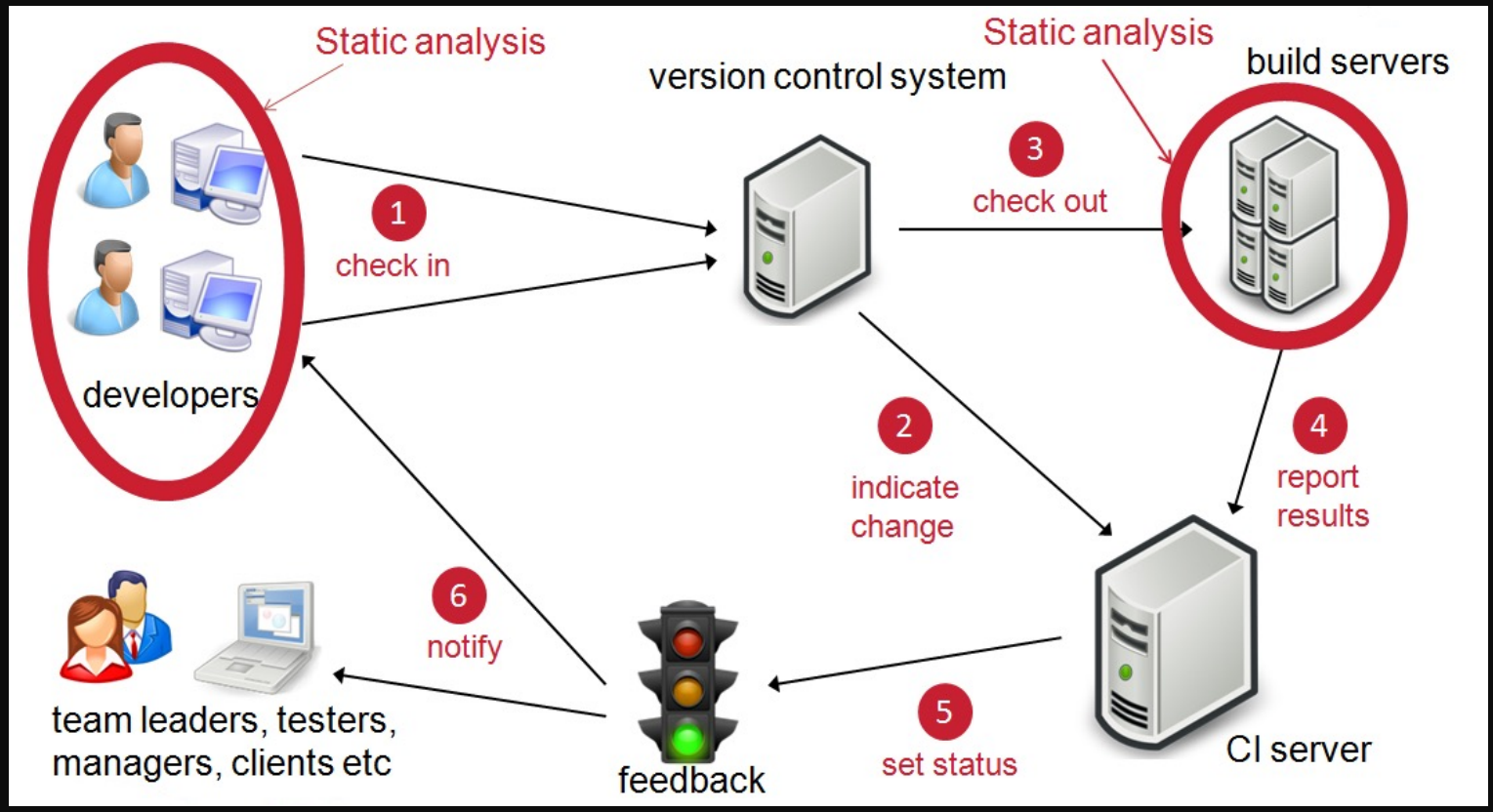
Release manager merges the code with the main branch and pushes out the new release



[image from http://agilelucero.com/extreme-programming/what-is-continuous-integration-ci/]

Continuous Integration (CI)

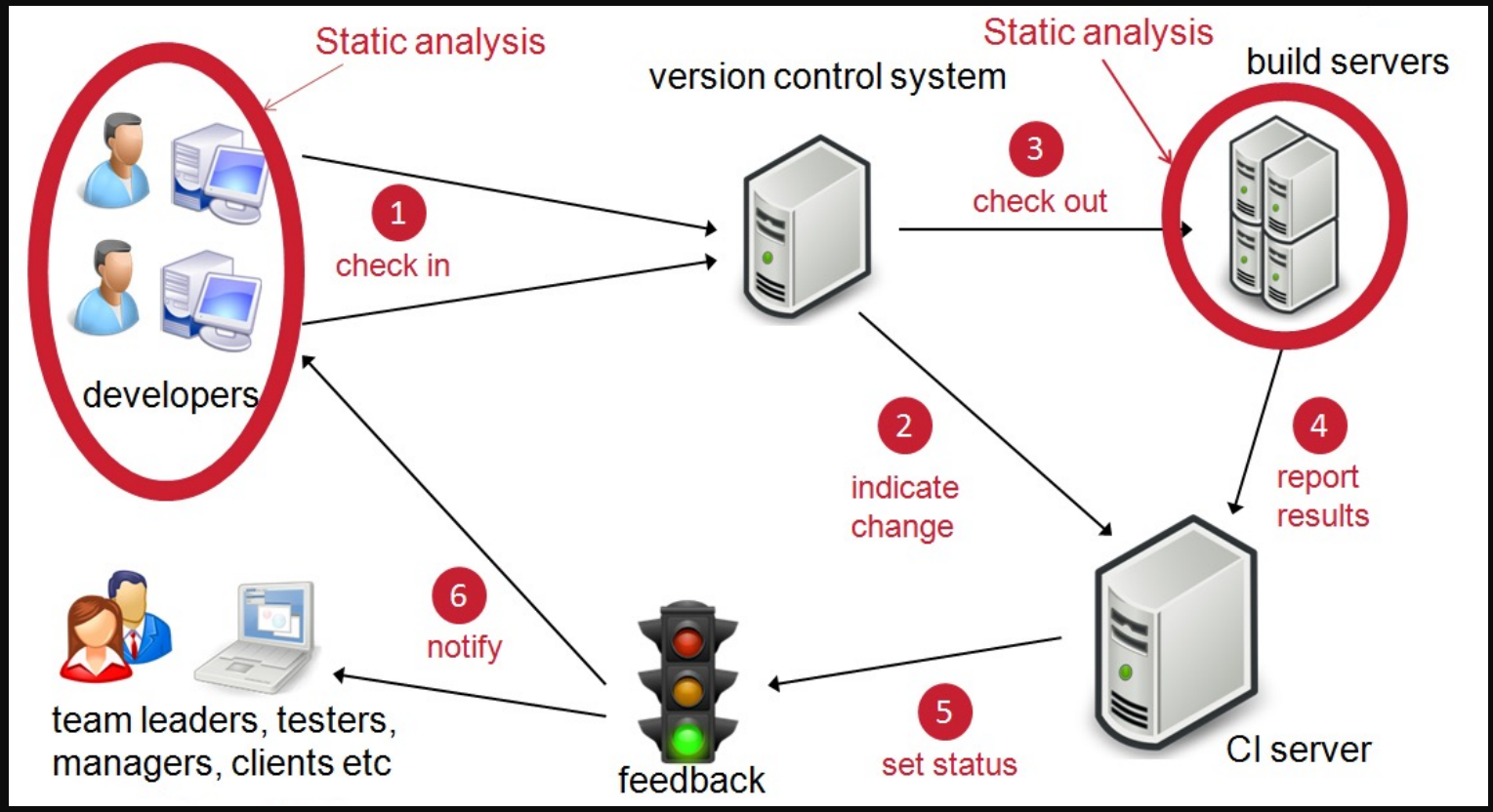
CI system monitors the version control system for changes and then launches the build after getting the source code from the repo



[image from http://agilelucero.com/extreme-programming/what-is-continuous-integration-ci/]

Continuous Integration (CI)

CI server runs unit tests and final tests to check validity and quality of the product, report status

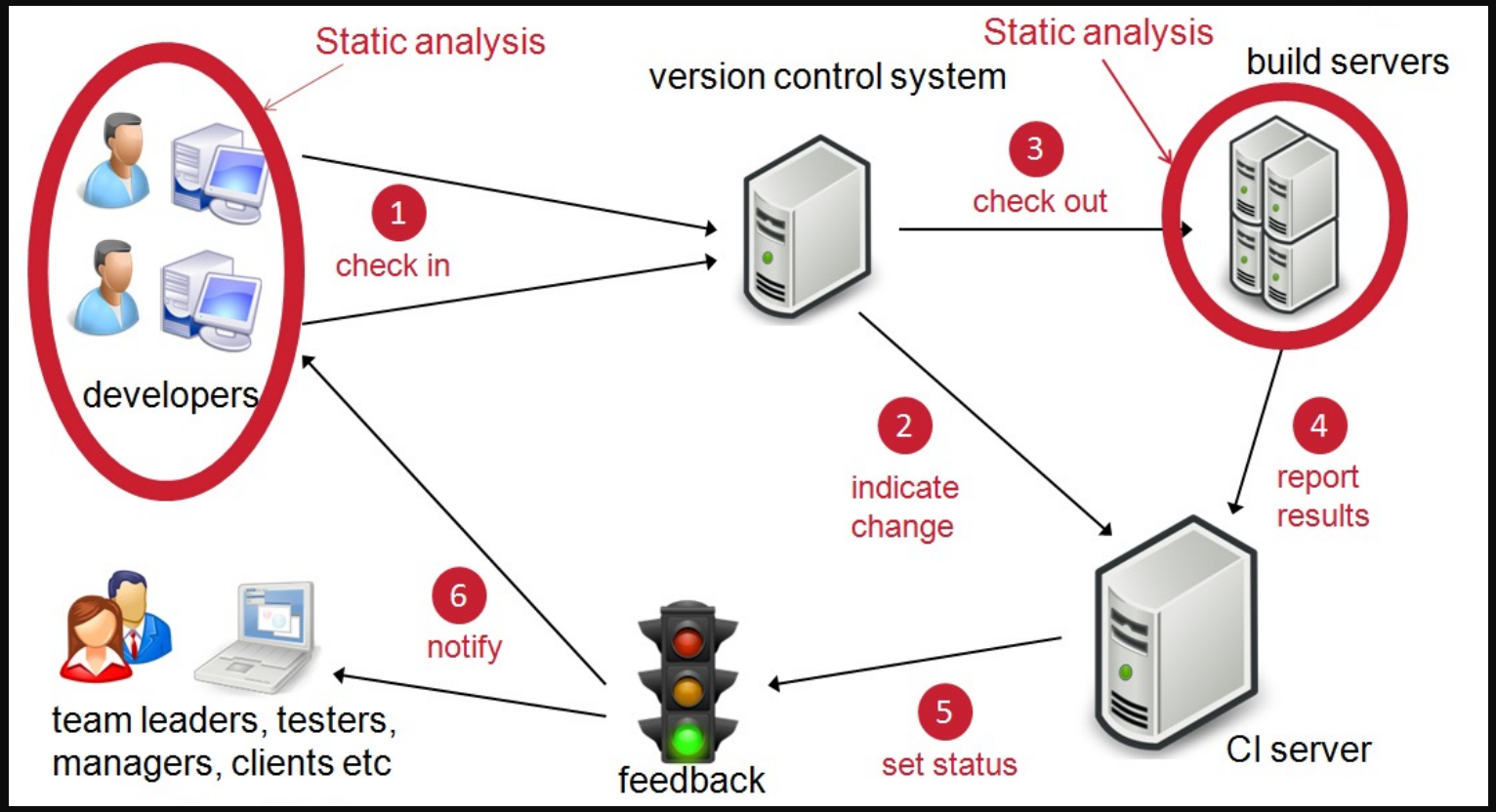


[image from http://agilelucero.com/extreme-programming/what-is-continuous-integration-ci/]

Continuous Integration (CI)

If success, the same package is deployed for acceptance testing and is then deployed on the production server

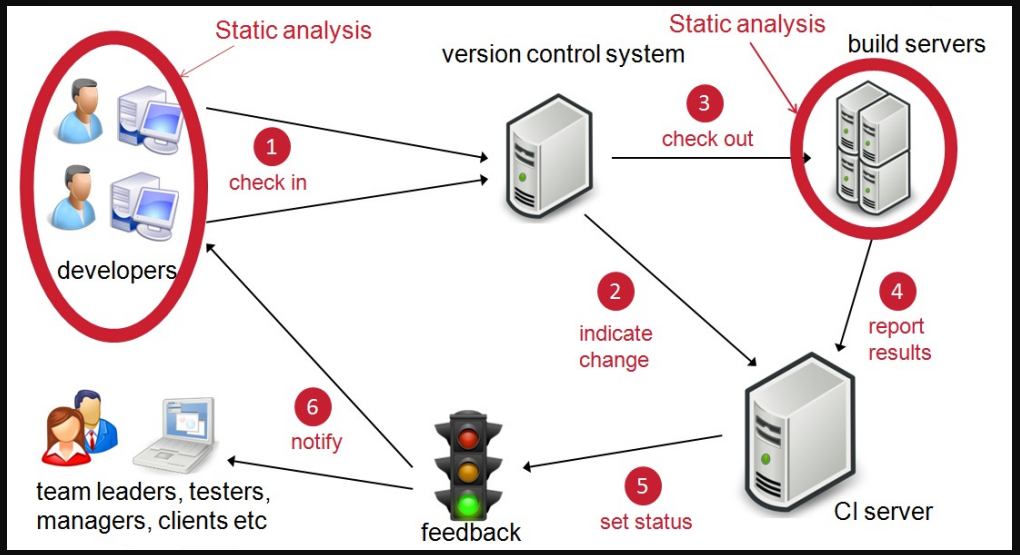
Otherwise, developers fix the code



[image from http://agilelucero.com/extreme-programming/what-is-continuous-integration-ci/]

Continuous Integration (CI)

A Continuous integration server rebuilds the system, returns, and re-verifies tests whenever any update is checked into the repo



Mistakes are caught earlier

Other developers are aware of changes early

The rebuild and re-verify must happen as soon as possible (tests need to execute quickly)

A continuous integration server doesn't just run tests, it decides if a modified system is still correct

[image from http://agilelucero.com/extreme-programming/what-is-continuous-integration-ci/]

CI: Things to Do

- Maintain a single source code
- Automate the build
- Keep the build fast and make it self-testing
- Every commit has to be built on the integration machine
- CI server completely informs the responsive teams of each successful build and alerts the team in case of any failure
- The team must ensure that the issue is fixed at the earliest

CI: Key Testing Areas

- Regression testing – ensure changes does not break the app
  - Run in the background
  - Provide regular feedback to minimize regression defects
- Performance testing – ensure baseline under normal conditions
  - Study the app for response time, identify changes in speed, reaction time, and app consistency
- Load testing – measure if the app can sustain the increased load
  - Measure response time when app is subjected to more than usual load
  - The load tests must begin small
- Scalability testing
  - Gauge the throughput, network, CPU memory usage, to reduce business risk
- End-to-end final testing – test the product in different scenarios

CI: Useful Resources

- Git (https://github.com/)
- Jenkins (https://jenkins.io/)
- Travis CI (http://travis-ci.org/)

Wrap-up

- More companies are putting testing first
- This can decrease cost and increase quality
- The definition of "correctness" becomes restricted but practical
- We embrace evolutionary design
- We use test harness as guardian
- Agile tests – most focus on "happy paths" and often miss
  - Confused-user paths
  - Creative-user paths
  - Malicious-user paths
- What's next?
  - Test-Driven Development (TDD)